

**Title:** Lecture - Numerical Methods, PHYS 777

**Speakers:** Erik Schnetter, Dustin Lang

**Collection/Series:** Numerical Methods (Core), PHYS 777-, January 6 - February 5, 2025

**Subject:** Other

**Date:** January 31, 2025 - 11:30 AM

**URL:** <https://pirsa.org/25010064>

Firefox File Edit View History Bookmarks Tools Window Help 09:33 MST 11:03:37:15 Fri Jan 31 11:33 AM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 17 hours ago (autosaved) Trusted Julia 1.8.5

File Edit View Insert Cell Kernel Widgets Help

Run

## Markov Chain Monte Carlo Lab

### Using Markov Chain Monte Carlo to infer the parameters of a simple model

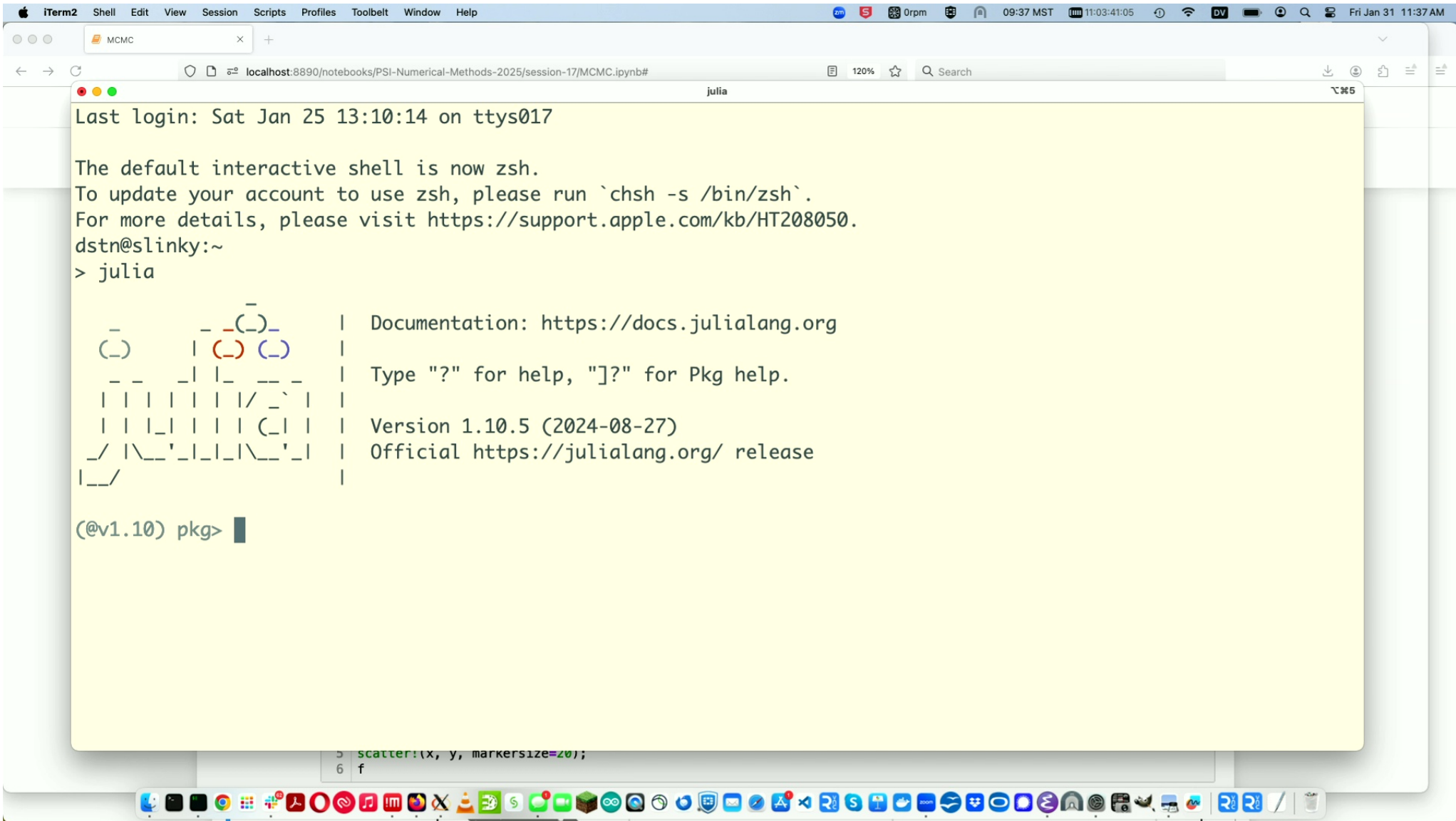
```
In [ ]: 1 ]add ForwardDiff
```

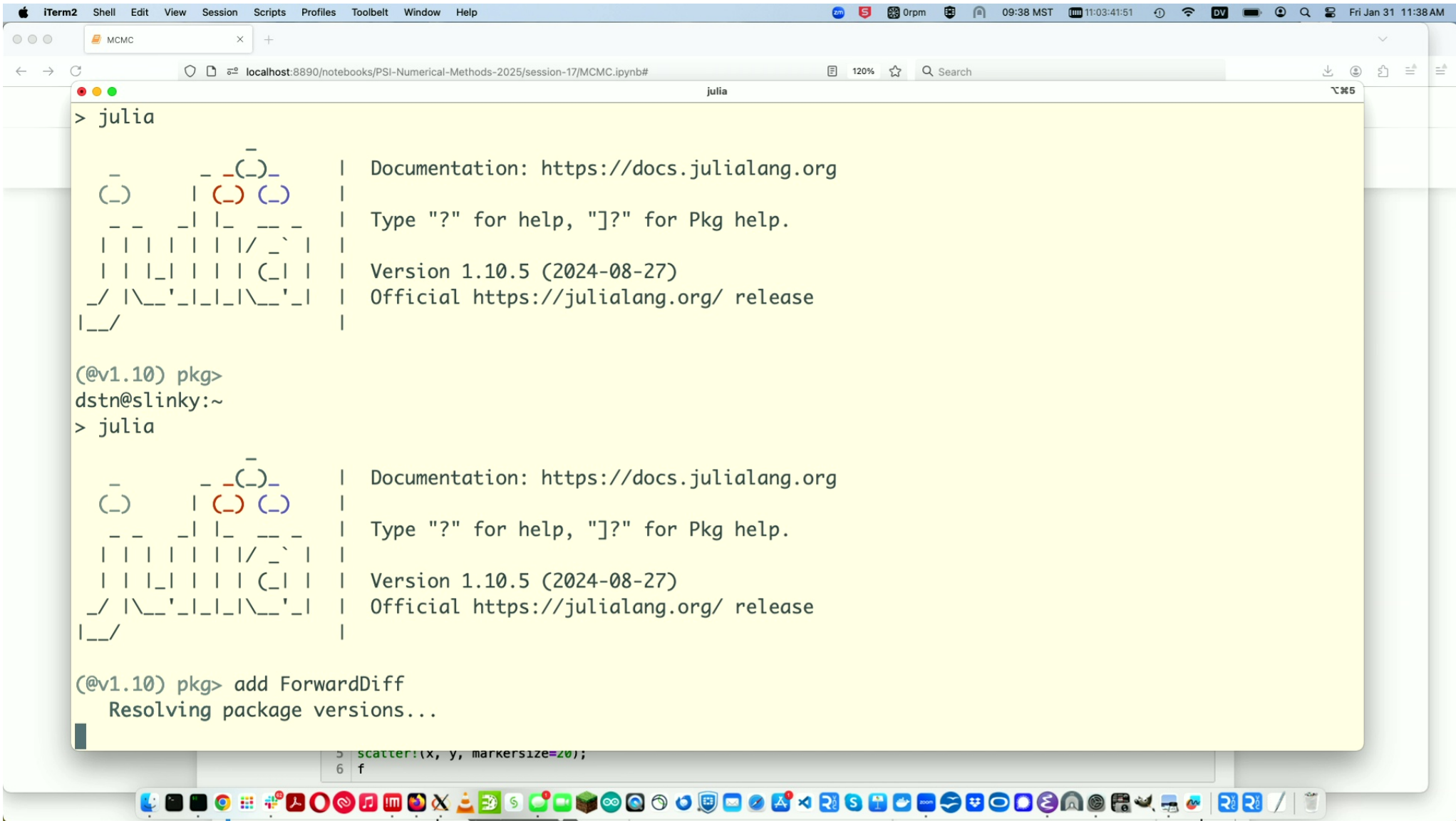
```
In [ ]: 1 ]add LogExpFunctions
```

```
In [ ]: 1 using CairoMakie
2 using Optim
3 using ForwardDiff
4 using LogExpFunctions
```

```
In [ ]: 1 # Example data set from arxiv:1008.4686, table 1 (https://arxiv.org/abs/1008.4686)
2 # You can also refer to that paper for more background, equations, etc.
3 alldata = [201. 592 61; 244 401 25; 47 583 38; 287 402 15; 203 495 21; 58 173 15; 210 479 27;
4           202 504 14; 198 510 30; 158 416 16; 165 393 14; 201 442 25; 157 317 52; 131 311 16;
5           166 400 34; 160 337 31; 186 423 42; 125 334 26; 218 533 16; 146 344 22 ]
6 # The first 5 data points are outliers; for the first part we'll just use the "good" data points
7 x = alldata[6:end, 1]
8 y = alldata[6:end, 2]
9 # this is the standard deviation (uncertainty) on the y measurements, also known as |sigma_i
10 yerr = alldata[6:end, 3];
```

```
In [ ]: 1 # To start, let's have a look at our data set.
2 f = Figure()
3 Axis(f[1, 1], xlabel="x", ylabel="y")
4 errorbars!(x, y, yerr);
5 scatter!(x, y, markersize=20);
6 f
```





Firefox File Edit View History Bookmarks Tools Window Help 09:39 MST 11:03:42:53 Fri Jan 31 11:39 AM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 17 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.8.5

## Markov Chain Monte Carlo Lab

### Using Markov Chain Monte Carlo to infer the parameters of a simple model


```
In [*]: 1 ]add ForwardDiff
```

```
In [ ]: 1 ]add LogExpFunctions|
```

```
In [ ]: 1 using CairoMakie
2 using Optim
3 using ForwardDiff
4 using LogExpFunctions
```

```
In [ ]: 1 # Example data set from arxiv:1008.4686, table 1 (https://arxiv.org/abs/1008.4686)
2 # You can also refer to that paper for more background, equations, etc.
3 alldata = [201. 592 61; 244 401 25; 47 583 38; 287 402 15; 203 495 21; 58 173 15; 210 479 27;
4           202 504 14; 198 510 30; 158 416 16; 165 393 14; 201 442 25; 157 317 52; 131 311 16;
5           166 400 34; 160 337 31; 186 423 42; 125 334 26; 218 533 16; 146 344 22 ]
6 # The first 5 data points are outliers; for the first part we'll just use the "good" data points
7 x = alldata[6:end, 1]
8 y = alldata[6:end, 2]
9 # this is the standard deviation (uncertainty) on the y measurements, also known as |sigma_i
10 yerr = alldata[6:end, 3];
```

```
In [ ]: 1 # To start, let's have a look at our data set.
2 f = Figure()
3 Axis(f[1, 1], xlabel="x", ylabel="y")
4 errorbars!(x, y, yerr);
5 scatter!(x, y, markersize=20);
6 f
```



Firefox File Edit View History Bookmarks Tools Window Help 09:41 MST 11:03:45:00 Fri Jan 31 11:41 AM

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb#

jupyter MCMC Last Checkpoint: 17 hours ago (unsaved changes) Logout

No kernel Trusted Julia 1.10.5

## Markov Chain Monte Carlo Lab

### Using Markov Chain Monte Carlo to infer the parameters of a simple model


```
In [1]: 1 |add ForwardDiff
Updating registry at `~/julia/registries/General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
No Changes to `~/julia/environments/v1.8/Project.toml`
No Changes to `~/julia/environments/v1.8/Manifest.toml`

In [2]: 1 |add LogExpFunctions
Resolving package versions...
No Changes to `~/julia/environments/v1.8/Project.toml`
No Changes to `~/julia/environments/v1.8/Manifest.toml`

In [3]: 1 using CairoMakie
2 using Optim
3 using ForwardDiff
4 using LogExpFunctions

ArgumentError: Package CairoMakie not found in current path.
- Run `import Pkg; Pkg.add("CairoMakie")` to install the CairoMakie package.

Stacktrace:
 [1] macro expansion
   @ ./loading.jl:1163 [inlined]
 [2] macro expansion
   @ ./lock.jl:223 [inlined]
 [3] require(into::Module, mod::Symbol)
```



Firefox File Edit View History Bookmarks Tools Window Help 09:41 MST 11:03:45:35 Fri Jan 31 11:41 AM

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 17 hours ago (unsaved changes) Logout

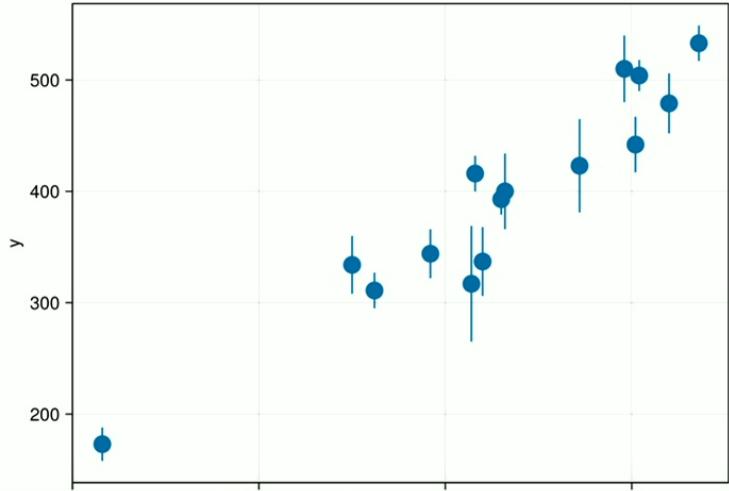
File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

```
4      202 504 14; 198 510 30; 158 416 16; 165 393 14; 201 442 25; 157 317 52; 131 311 16;
5      166 400 34; 160 337 31; 186 423 42; 125 334 26; 218 533 16; 146 344 22 ]
6      # The first 5 data points are outliers; for the first part we'll just use the "good" data points
7      x = alldata[6:end, 1]
8      y = alldata[6:end, 2]
9      # this is the standard deviation (uncertainty) on the y measurements, also known as |sigma_i
10     yerr = alldata[6:end, 3];
```

In [3]:

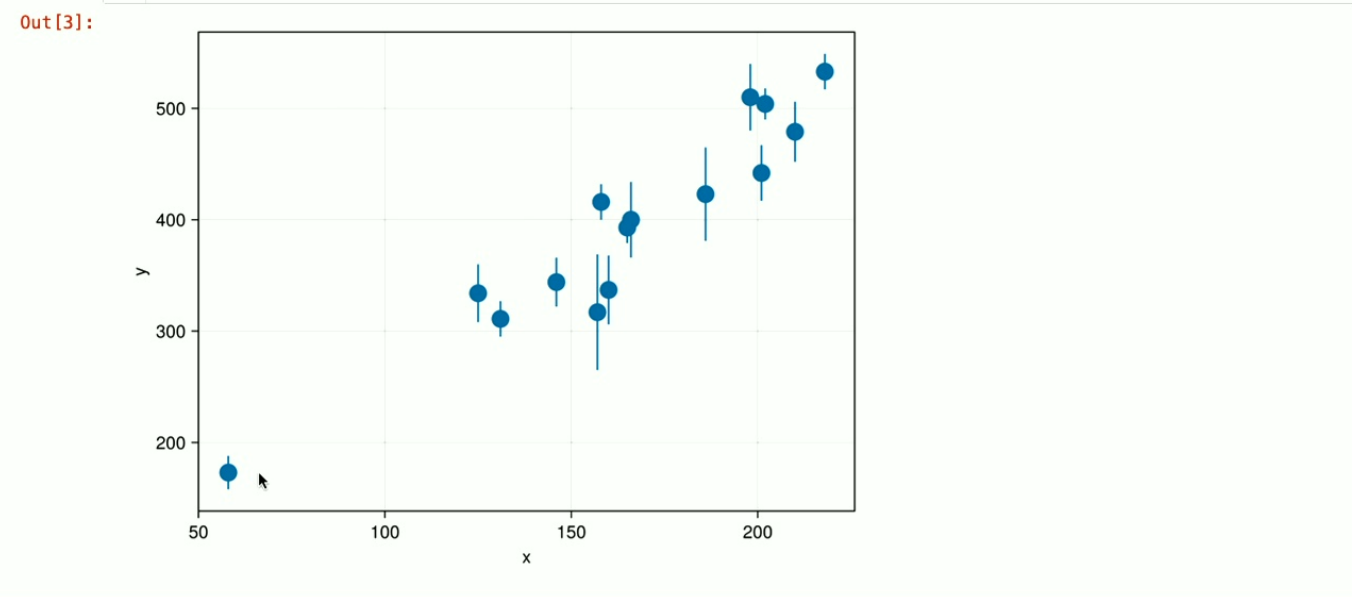
```
1      # To start, let's have a look at our data set.
2      f = Figure()
3      Axis(f[1, 1], xlabel="x", ylabel="y")
4      errorbars!(x, y, yerr);
5      scatter!(x, y, markersize=20);
6      f
```

Out[3]:



x	y	yerr
100	170	0
130	330	20
140	310	15
150	340	25
160	320	30
165	410	20
170	390	25
180	420	30
200	420	35
205	500	30
210	510	35
215	440	30
220	480	35
240	530	40

```
5 scatter!(x, y, markersize=20);  
6 f
```



For this lab, we are going to imagine that there is some physical reason to believe there is a *linear* relationship between the quantities  $x$  and  $y$ , so our *generative model* of the process is that there is a "real" or "predicted" value  $y_{\text{pred}}$  given by  $y_{\text{pred}} = b + mx$ , for some parameters  $b$  and  $m$  that we will try to infer. Our *measurements*  $y$  are noisy measurements of  $y_{\text{pred}}$ , with additive Gaussian noise with known variance,  $\sigma_i$  for data point  $i$ . ( $\sigma$  is called *yerr* in this notebook.) This is a strong assumption about our data-collection method.

With those assumptions, we can write down the *likelihood* for a single measurement  $y_i$  given its corresponding  $x_i$  and  $\sigma_i$ , and straight-line model parameters  $b$  and  $m$ :





For this lab, we are going to imagine that there is some physical reason to believe there is a *linear* relationship between the quantities  $x$  and  $y$ , so our *generative model* of the process is that there is a "real" or "predicted" value  $y_{\text{pred}}$  given by  $y_{\text{pred}} = b + mx$ , for some parameters  $b$  and  $m$  that we will try to infer. Our *measurements*  $y$  are noisy measurements of  $y_{\text{pred}}$ , with additive Gaussian noise with known variance,  $\sigma_i$  for data point  $i$ . ( $\sigma$  is called *yerr* in this notebook.) This is a strong assumption about our data-collection method.

With those assumptions, we can write down the *likelihood* for a single measurement  $y_i$  given its corresponding  $x_i$  and  $\sigma_i$ , and straight-line model parameters  $b$  and  $m$ :

$$y_{\text{pred},i} = b + mx_i$$
$$p(y_i | m, b) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - y_{\text{pred},i})^2}{2\sigma_i^2}\right)$$

We are making a number of simplifying assumptions here:

- there are no uncertainties on the  $x$  values
- there are additive Gaussian measurement uncertainties on the  $y$  values with known standard deviations  $\sigma$
- the data points are statistically independent

Since we are assuming the  $x$  and  $\sigma$  values are perfectly known, we will treat them as *constants* rather than *data* in the probability equations.

In practice, it is usually preferable to work in log-probabilities rather than linear probabilities, because the probability values can be very small, and if we're not careful we can hit a numerical issue called *underflow*, where the numbers become so small that they can't be represented in standard floating-point numerical representation.

If the data points are statistically independent, then the likelihood of the whole collection of data points  $y = \{y_i\}$  is the *product* of their individual

Firefox File Edit View History Bookmarks Tools Window Help 09:46 MST 11:03:50:03 Fri Jan 31 11:46 AM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 17 hours ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

Run

We are making a number of simplifying assumptions here.

- there are no uncertainties on the  $x$  values
- there are additive Gaussian measurement uncertainties on the  $y$  values with known standard deviations  $\sigma$
- the data points are statistically independent

Since we are assuming the  $x$  and  $\sigma$  values are perfectly known, we will treat them as *constants* rather than *data* in the probability equations.

In practice, it is usually preferable to work in log-probabilities rather than linear probabilities, because the probability values can be very small, and if we're not careful we can hit a numerical issue called *underflow*, where the numbers become so small that they can't be represented in standard floating-point numerical representation.

If the data points are statistically independent, then the likelihood of the whole collection of data points  $y = \{y_i\}$  is the *product* of their individual likelihoods:

$$p(y|m, b) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y-y_{\text{pred},i})^2}{2\sigma_i^2}\right)$$

and taking the log,

$$\log p(y|m, b) = \sum_i \log\left(\frac{1}{\sqrt{2\pi}\sigma_i}\right) - \frac{(y-y_{\text{pred},i})^2}{2\sigma_i^2}$$

The first thing we will do is implement that log-likelihood function!

```
In [ ]: 1 function log_likelihood_one(params, x, y, yerr)
2         """This function computes the log-likelihood of a data set with coordinates
3         (x_i,y_i) and Gaussian uncertainties on y_i of yerr_i (aka sigma_i)
4
5         The model is a straight line, so the model's predicted y values are
6         y_pred_i = b + m x_i.
7
8         params = (b,m) are the parameters (scalars)
9         x,y,yerr are arrays (aka vectors)
10
11        Return value is a scalar log-likelihood.
12        """
13        # unpack the parameters
```

and taking the log,

$$\log p(y|m, b) = \sum_i \log\left(\frac{1}{\sqrt{2\pi}\sigma_i}\right) - \frac{(y - y_{\text{pred},i})^2}{2\sigma_i^2}$$

The first thing we will do is implement that log-likelihood function!

```
In [ ]: 1 function log_likelihood_one(params, x, y, yerr)
2         """This function computes the log-likelihood of a data set with coordinates
3         (x_i,y_i) and Gaussian uncertainties on y_i of yerr_i (aka sigma_i)
4
5         The model is a straight line, so the model's predicted y values are
6         y_pred_i = b + m x_i.
7
8         params = (b,m) are the parameters (scalars)
9         x,y,yerr are arrays (aka vectors)
10
11        Return value is a scalar log-likelihood.
12        """
13        # unpack the parameters
14        b,m = params
15        # compute the vector y_pred, the model predictions for the y measurements
16        y_pred = b .+ m .* x
17        # compute the log-likelihoods for the individual data points
18        # (the quantity inside the sum in the text above)
19        ### FILL IN CODE HERE! Implement the log-likelihood function from the text above!
20        loglikes = #log.( ... ) -. 0.5 .* ().^ ./ ().^2
21        # the log-likelihood for the whole vector of measurements is the sum of individual log-likelihoods
22        loglike = sum(loglikes)
23        return loglike
24    end;
```

## Maximum likelihood

Before we start experimenting Markov Chain Monte Carlo, let's use a stock optimizer routine from Julia's *Optim* package. The optimizer will allow us to find the *maximum likelihood* paramaters *b* and *m*.

Firefox File Edit View History Bookmarks Tools Window Help 09:55 MST 11:03:59:33 Fri Jan 31 11:55 AM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 17 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

Run Code

## Maximum likelihood

Before we start experimenting Markov Chain Monte Carlo, let's use a stock optimizer routine from Julia's *Optim* package. The optimizer will allow us to find the *maximum likelihood* parameters  $b$  and  $m$ .

Since the optimizer wants to *minimize* a function but we want to *maximize* the log-likelihood, we need to add a negative sign...

```
In [ ]: 1 # The optimizer we're using here requires an initial guess. This log-likelihood happens
2 # to be pretty simple, so we don't need to work very hard to give it a good initial guess!
3 initial_params = [0., 0.]
4 # The "args" parameter here gets passed to the neg_ll_one function (after the parameters)
5 result = optimize(p -> -log_likelihood_one(p, x, y, yerr), initial_params)
```

```
In [ ]: 1 b_ml,m_ml = Optim.minimizer(result)
```

Previously, we used the jack-knife routine to estimate variances on our parameters. We can also compute the second derivative of the log-likelihood function, at the peak, to get an estimate of uncertainties on the parameters. This is related to the Fisher information matrix, if you've ever encountered that.

```
In [ ]: 1 # Don't worry about understanding this!
2 invhess = inv(ForwardDiff.hessian(p -> -log_likelihood_one(p, x, y, yerr), [b_ml,m_ml]))
```

```
In [ ]: 1 # The optimizer gives us the parameters that maximize the log-likelihood, along with an estimate of the uncertainty
2 # To start, let's have a look at our data set.
3 f = Figure()
4 Axis(f[1, 1], xlabel="B", ylabel="M")
5 errorbars!(x, y, yerr);
6 scatter!(x, y, markersize=20);
7 xx = LinRange(50, 250, 50)
8 lines!(xx, b_ml .+ m_ml .* xx)
9 # Draw a sampling of B,M parameter values that are consistent with the fit,
10 # using the estimated inverse-Hessian matrix (parameter covariance)
11 using LinearAlgebra
12 # use the svd to draw multivariate random normal samples!
13 S = svd(invhess)
14 BM = [S.U * Diagonal(sqrt.(S.S)) * randn(2) for i in 1:10]
```

Firefox File Edit View History Bookmarks Tools Window Help 09:57 MST 11:04:00:56 Fri Jan 31 11:57 AM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 17 hours ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

## Maximum likelihood

Before we start experimenting Markov Chain Monte Carlo, let's use a stock optimizer routine from Julia's *Optim* package. The optimizer will allow us to find the *maximum likelihood* parameters *b* and *m*.

Since the optimizer wants to *minimize* a function but we want to *maximize* the log-likelihood, we need to add a negative sign...

```
In [13]: 1 # The optimizer we're using here requires an initial guess. This log-likelihood happens
2 # to be pretty simple, so we don't need to work very hard to give it a good initial guess!
3 initial_params = [0., 0.]
4 # The "args" parameter here gets passed to the neg_ll_one function (after the parameters)
5 result = optimize(p -> -log_likelihood_one(p, x, y, verr), initial_params)
```

```
Out[13]: * Status: success
* Candidate solution
  Final objective value: 7.029347e+01
* Found with
  Algorithm: Nelder-Mead
* Convergence measures
   $\sqrt{(\Sigma(y_i - \bar{y})^2)/n} \leq 1.0e-08$ 
* Work counters
  Seconds run: 0 (vs limit Inf)
  Iterations: 70
  f(x) calls: 135
```

```
In [ ]: 1 b_ml,m_ml = Optim.minimizer(result)
```

Previously, we used the jack-knife routine to estimate variances on our parameters. We can also compute the second derivative of the log-likelihood function, at the peak, to get an estimate of uncertainties on the parameters. This is related to the Fisher information matrix, if you've ever encountered that.

```
In [ ]: 1 # Don't worry about understanding this!
2 invhess = inv(ForwardDiff.hessian(p -> -log_likelihood_one(p, x, y, verr), [b_ml,m_ml]))
```

Firefox File Edit View History Bookmarks Tools Window Help 09:58 MST 11:04:02:05 Fri Jan 31 11:58 AM

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb#

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

Run Code

Since the optimizer wants to *minimize* a function but we want to *maximize* the log-likelihood, we need to add a negative sign...

```
In [14]: 1 # The optimizer we're using here requires an initial guess. This log-likelihood happens
2 # to be pretty simple, so we don't need to work very hard to give it a good initial guess!
3 initial_params = [0., 0.]
4 # The "args" parameter here gets passed to the neg_ll_one function (after the parameters)
5 result = optimize(p -> -log_likelihood_one(p, x, y, yerr), initial_params)
```

```
Out[14]: * Status: success
* Candidate solution
  Final objective value: 7.029347e+01
* Found with
  Algorithm: Nelder-Mead
* Convergence measures
   $\sqrt{(\Sigma(y_i - \bar{y})^2)/n} \leq 1.0e-08$ 
* Work counters
  Seconds run: 0 (vs limit Inf)
  Iterations: 70
  f(x) calls: 135
```

```
In [ ]: 1 # To start, let's have a look at our data set.
2 f = Figure()
3 Axis(f[1, 1], xlabel="x", ylabel="y")
4 errorbars!(x, y, yerr);
5 scatter!(x, y, markersize=20);
6 lines!(x ->)
7 f
```

```
In [ ]: 1
```

```
In [ ]: 1
```

Firefox File Edit View History Bookmarks Tools Window Help 12:00 PM

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb#

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

```
In [19]: 1 # To start, let's have a look at our data set.
2 f = Figure()
3 Axis(f[1, 1], xlabel="x", ylabel="y")
4 errorbars(x, y, yerr);
5 scatter!(x, y, markersize=20);
6 xx = range(50,200)
7 lines!(xx, 150 .* 2. .* xx)
8 f
```

Out [19]:

In [ ]: 1

Since the optimizer wants to *minimize* a function but we want to *maximize* the log-likelihood, we need to add a negative sign...

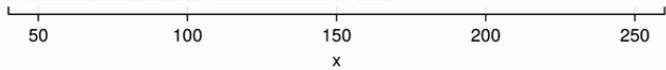
```
In [14]: 1 # The optimizer we're using here requires an initial guess. This log-likelihood happens
2 # to be pretty simple, so we don't need to work very hard to give it a good initial guess!
3 initial_params = [0., 0.]
4 # The "args" parameter here gets passed to the neg_ll_one function (after the parameters)
5 result = optimize(p -> -log_likelihood_one(p, x, y, yerr), initial_params)
```

```
Out[14]: * Status: success
* Candidate solution
  Final objective value: 7.029347e+01
* Found with
  Algorithm: Nelder-Mead
* Convergence measures
   $\sqrt{(\Sigma(y_i - \bar{y})^2)/n} \leq 1.0e-08$ 
* Work counters
  Seconds run: 0 (vs limit Inf)
  Iterations: 70
  f(x) calls: 135
```

```
In [21]: 1 optimize(log_likelihood_one, initial_params)
```

```
MethodError: no method matching log_likelihood_one(::Vector{Float64})
Closest candidates are:
  log_likelihood_one(::Any, ::Any, ::Any, ::Any)
    @ Main In[5]:1
Stacktrace:
 [1] value!!(obj::NonDifferentiable{Float64, Vector{Float64}}, x::Vector{Float64})
    @ NLSolversBase ~/.julia/packages/NLSolversBase/kavn7/src/interface.jl:9
 [2] initial_state(method::NelderMead{Optim.AffineSimplexer, Optim.AdaptiveParameters}, options::Optim.Options{Floa
```





Previously, we used the jack-knife routine to estimate variances on our parameters. We can also compute the second derivative of the log-likelihood function, at the peak, to get an estimate of uncertainties on the parameters. This is related to the Fisher information matrix, if you've ever encountered that.

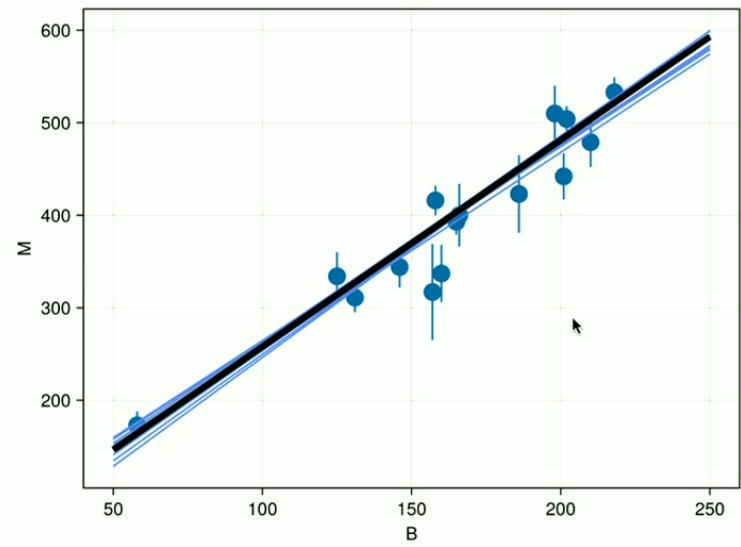
```
In [*]: 1 # Don't worry about understanding this!
        2 invhess = inv(ForwardDiff.hessian(p -> -log_likelihood_one(p, x, y, yerr), [b_ml,m_ml]))
```

```
In [ ]: 1 # The optimizer gives us the parameters that maximize the log-likelihood, along with an estimate of the uncertainty
        2 # To start, let's have a look at our data set.
        3 f = Figure()
        4 Axis(f[1, 1], xlabel="B", ylabel="M")
        5 errorbars!(x, y, yerr);
        6 scatter!(x, y, markersize=20);
        7 xx = LinRange(50, 250, 50)
        8 lines!(xx, b_ml .+ m_ml .* xx)
        9 # Draw a sampling of B,M parameter values that are consistent with the fit,
        10 # using the estimated inverse-Hessian matrix (parameter covariance)
        11 using LinearAlgebra
        12 # use the svd to draw multivariate random normal samples!
        13 S = svd(invhess)
        14 BM = [S.U * Diagonal(sqrt.(S.S)) * randn(2) for i in 1:10]
        15 for (db,dm) in BM
        16     lines!(xx, (b_ml .+ db) .+ (m_ml .+ dm) .* xx, color=:cornflowerblue)
        17 end
        18 lines!(xx, b_ml .+ m_ml .* xx, color=:black, linewidth=5)
        19 f
```

```
In [ ]: 1 # You can also plot the ellipse showing the constraints in B,M space by manipulating hess_inv
        2 # (don't worry about understanding this math)
        3 SS = S.U * Diagonal(sqrt.(S.S))
        4 th = LinRange(0., 2π, 200)
        5 xx = sin.(th)
        6 yy = cos.(th)
```

```
9 # Draw a sampling of B,M parameter values that are consistent with the fit,  
10 # using the estimated inverse-Hessian matrix (parameter covariance)  
11 using LinearAlgebra  
12 # use the svd to draw multivariate random normal samples!  
13 S = svd(invhess)  
14 BM = [S.U * Diagonal(sqrt.(S.S)) * randn(2) for i in 1:10]  
15 for (db,dm) in BM  
16     lines!(xx, (b_ml .+ db) .+ (m_ml .+ dm) .* xx, color=:cornflowerblue)  
17 end  
18 lines!(xx, b_ml .+ m_ml .* xx, color=:black, linewidth=5)  
19 f
```

Out [26]:



Firefox File Edit View History Bookmarks Tools Window Help

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb#

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Julia 1.10.5

## Markov Chain Monte Carlo

Next, let's implement the Markov Chain Monte Carlo algorithm.

The MCMC algorithm moves a "particle" or "sample" or "walker" randomly around the particle space, by first proposing a move, and then using the relative likelihoods of the current and proposed positions to decide whether to accept or reject the move.

```
In [ ]: 1 function mcmc(logprob_func,
2           propose_func,
3           initial_pos, nsteps)
4         """
5         MCMC: Markov Chain Monte Carlo. Draw samples from the *logprob_func* probability distribution,
6         using proposed moves generated by the function *propose_func*.
7
8         * logprob_func: a function that returns the log-probability at a given value of parameters.
9           It will get called like this:
10            lnp = logprob_func(params, logprob_args)
11         * propose_func: a function that proposes to jump to a new point in parameter space.
12           It will get called like this:
13            p_new = propose_func(p, propose_args)
14         * initial_pos: initial position in parameter space (list/array)
15         * nsteps: integer number of MCMC steps to take
16
17         Returns (chain, faccept)
18         * chain: size Nsteps x P, MCMC samples
19         * faccept: float: fraction of proposed jumps that were accepted
20         """
21         p = initial_pos
22         logprob = logprob_func(p)
23         chain = zeros(Float64, (nsteps, length(p)))
24         naccept = 0
25         for i in 1:nsteps
26             # propose a new position in parameter space
27             ### FILL IN CODE HERE -- propose a jump to a new place in param space
28             p_new = #...
29             # compute probability at new position
30             ### FILL IN CODE HERE
```

Firefox File Edit View History Bookmarks Tools Window Help 10:10 MST 11:04:14:28 Fri Jan 31 12:10 PM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 18 hours ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

```

9
10     lnp = logprob_func(params, logprob_args)
11     * propose_func: a function that proposes to jump to a new point in parameter space.
12     It will get called like this:
13     p_new = propose_func(p, propose_args)
14     * initial_pos: initial position in parameter space (list/array)
15     * nsteps: integer number of MCMC steps to take
16
17     Returns (chain, faccept)
18     * chain: size Nsteps x P, MCMC samples
19     * faccept: float: fraction of proposed jumps that were accepted
20     """
21     p = initial_pos
22     logprob = logprob_func(p)
23     chain = zeros(Float64, (nsteps, length(p)))
24     naccept = 0
25     for i in 1:nsteps
26         # propose a new position in parameter space
27         ### FILL IN CODE HERE -- propose a jump to a new place in param space
28         p_new = #...
29         # compute probability at new position
30         ### FILL IN CODE HERE
31         logprob_new = #...
32         # decide whether to jump to the new position
33         ### FILL IN CODE HERE!!!
34         if exp( ..... ) > rand()
35             p = p_new
36             logprob = logprob_new
37             naccept += 1
38         end
39         # save the position
40         chain[i,:] = p
41     end
42     return chain, naccept/nsteps
43 end;

```

We'll use a Gaussian (without covariance between the parameters) for our proposal distribution:

To [ ] function propose\_gaussian(p, sigma)

Firefox File Edit View History Bookmarks Tools Window Help

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb#

120% Search

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

```
9
10     lnp = logprob_func(params, logprob_args)
11     * propose_func: a function that proposes to jump to a new point in parameter space.
12     It will get called like this:
13     p_new = propose_func(p, propose_args)
14     * initial_pos: initial position in parameter space (list/array)
15     * nsteps: integer number of MCMC steps to take
16
17     Returns (chain, faccept)
18     * chain: size Nsteps x P, MCMC samples
19     * faccept: float: fraction of proposed jumps that were accepted
20     """
21     p = initial_pos
22     logprob = logprob_func(p)
23     chain = zeros(Float64, (nsteps, length(p)))
24     naccept = 0
25     for i in 1:nsteps
26         # propose a new position in parameter space
27         p_new = propose_func(p)
28         # compute probability at new position
29         logprob_new = logprob_func(p_new)
30         # decide whether to jump to the new position
31         if p_new > p
32             # jump there
33         else
34             # maybe jump there
35             ratio =
36 #             if exp( .... ) > rand()
37 #                 p = p_new
38 #                 logprob = logprob_new
39 #                 naccept += 1
40 #             end
41             # save the position
42             chain[i,:] = p
43         end
44     return chain, naccept/nsteps
45 end;
```

We'll use a Gaussian (without covariance between the parameters) for our proposal distribution:

Firefox File Edit View History Bookmarks Tools Window Help 10:14 MST 11:04:18:04 Fri Jan 31 12:14 PM

MCMC

localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 18 hours ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Notebook saved Trusted Julia 1.10.5

```
9
10     lnp = logprob_func(params, logprob_args)
11     * propose_func: a function that proposes to jump to a new point in parameter space.
12     It will get called like this:
13     p_new = propose_func(p, propose_args)
14     * initial_pos: initial position in parameter space (list/array)
15     * nsteps: integer number of MCMC steps to take
16
17     Returns (chain, faccept)
18     * chain: size Nsteps x P, MCMC samples
19     * faccept: float: fraction of proposed jumps that were accepted
20     """
21     p = initial_pos
22     logprob = logprob_func(p)
23     chain = zeros(Float64, (nsteps, length(p)))
24     naccept = 0
25     for i in 1:nsteps
26         # propose a new position in parameter space
27         p_new = propose_func(p)
28         # compute probability at new position
29         logprob_new = logprob_func(p_new)
30         # decide whether to jump to the new position
31         if logprob_new > logprob
32             # jump there
33             p = p_new
34             logprob = logprob_new
35             naccept += 1
36         else
37             # maybe jump there
38             ratio = exp(logprob_new - logprob)
39             if ratio > rand()
40                 # jump there
41             else
42                 # stay where we are
43             end
44         end
45     end
46     # if exp( .... ) > rand()
47     #     p = p_new
48     #     logprob = logprob_new
49     #     naccept += 1
```

```
45     # stay where we are
46     end
47     end
48     # save the position
49     chain[i,:] = p
50 end
51 return chain, naccept/nsteps
52 end;
```

We'll use a Gaussian (without covariance between the parameters) for our proposal distribution:

```
In [ ]: 1 function propose_gaussian(p, stdevs)
2       """
3       A Gaussian proposal distribution for mcmc.
4       *p*: the point in parameter space to jump from
5       *stdevs*: standard deviations for each dimension in the parameter space.
6       """
7       return p .+ randn(length(p)) .* stdevs
8     end;
```

Now, we defined our log-likelihood function above, but when using MCMC for Bayesian inference, we need to pass it a log-posterior function. That is, we must include the log-prior for the parameters. It is very common to see "uninformative" or "flat" priors used; in fact, it's not uncommon to see the log-prior just set to zero, as below, which is, statistically speaking, a naughty thing to do, since that prior definitely isn't a proper probability distribution -- it isn't even bounded! But, it *feels* like we haven't imposed our *subjective* prior beliefs on the inference, which is why people often do it. But you can't avoid subjectivity---if you change the parameterization, for example, a flat prior becomes non-flat!

```
In [ ]: 1 function log_posterior_one(params, x, y, err)
2       loglike = log_likelihood_one(params, x, y, yerr)
3       # Improper, flat priors on params!
4       logprior = 0.
5       return loglike + logprior
6     end;
```

```
In [ ]: 1 # initial B,M
2 initial_pos = [0., 1.0]
3 # proposal distribution: jump sizes for B,M
```

Firefox File Edit View History Bookmarks Tools Window Help 10:17 MST 11:04:21:10 Fri Jan 31 12:17 PM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

We use a Gaussian (without covariance between the parameters) for our proposal distribution.

```
In [29]: 1 function propose_gaussian(p, stdevs)
2         """
3         A Gaussian proposal distribution for mcmc.
4         *p*: the point in parameter space to jump from
5         *stdevs*: standard deviations for each dimension in the parameter space.
6         """
7         return p .+ randn(length(p)) .* stdevs
8     end;
```

Now, we defined our log-likelihood function above, but when using MCMC for Bayesian inference, we need to pass it a log-posterior function. That is, we must include the log-prior for the parameters. It is very common to see "uninformative" or "flat" priors used; in fact, it's not uncommon to see the log-prior just set to zero, as below, which is, statistically speaking, a naughty thing to do, since that prior definitely isn't a proper probability distribution -- it isn't even bounded! But, it *feels* like we haven't imposed our *subjective* prior beliefs on the inference, which is why people often do it. But you can't avoid subjectivity---if you change the parameterization, for example, a flat prior becomes non-flat!

```
In [ ]: 1 function log_posterior_one(params, x, y, err)
2         loglike = log_likelihood_one(params, x, y, yerr)
3         # Improper, flat priors on params!
4         logprior = 0.
5         return loglike + logprior
6     end;
```

```
In [ ]: 1 # initial B,M
2 initial_pos = [0., 1.0]
3 # proposal distribution: jump sizes for B,M
4 jump_sizes = [1., 0.1]
5 # Run MCMC!
6 chain,accept = mcmc(p -> log_posterior_one(p, x, y, err),
7                   p -> propose_gaussian(p, jump_sizes),
8                   initial_pos, 5000)
9 println("Fraction of moves accepted:", accept)
10 size(chain)
```

```
In [ ]: 1 # Plot the parameter values in the chain!
```





```
.....
7     return p .+ randn(length(p)) .* stdevs
8 end;
```

Now, we defined our log-likelihood function above, but when using MCMC for Bayesian inference, we need to pass it a log-posterior function. That is, we must include the log-prior for the parameters. It is very common to see "uninformative" or "flat" priors used; in fact, it's not uncommon to see the log-prior just set to zero, as below, which is, statistically speaking, a naughty thing to do, since that prior definitely isn't a proper probability distribution -- it isn't even bounded! But, it *feels* like we haven't imposed our *subjective* prior beliefs on the inference, which is why people often do it. But you can't avoid subjectivity--if you change the parameterization, for example, a flat prior becomes non-flat!

```
In [30]: 1 function log_posterior_one(params, x, y, err)
2         loglike = log_likelihood_one(params, x, y, yerr)
3         # Improper, flat priors on params!
4         logprior = 0.
5         return loglike + logprior
6     end;
```

```
In [ ]: 1 # initial B,M
2     initial_pos = [0., 1.0]
3     # proposal distribution: jump sizes for B,M
4     jump_sizes = [1., 0.1]
5     # Run MCMC!
6     chain,accept = mcmc(p -> log_posterior_one(p, x, y, err),
7                       p -> propose_gaussian(p, jump_sizes),
8                       initial_pos, 5000)
9     println("Fraction of moves accepted:", accept)
10    size(chain)
```

```
In [ ]: 1 # Plot the parameter values in the chain!
2     f = Figure()
3     Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC Samples")
4     scatter!(chain[:,1], chain[:,2], color=:grey)
5     lines!(ellipse_b, ellipse_m, color=:red)
6     f
```

Try re-running the MCMC cell above and re-plotting the results. Do the results look the same every time? Does that suggest anything to you about whether

Firefox File Edit View History Bookmarks Tools Window Help 10:21 MST 11:04:25:19 Fri Jan 31 12:21 PM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

```

2     loglike = log_likelihood_one(params, x, y, yerr)
3     # Improper, flat priors on params!
4     logprior = 0.
5     return loglike + logprior
6 end;

```

In [31]:

```

1 # initial B,M
2 initial_pos = [0., 1.0]
3 # proposal distribution: jump sizes for B,M
4 jump_sizes = [1., 0.1]
5 # Run MCMC!
6 chain,accept = mcmc(p -> log_posterior_one(p, x, y, err),
7                    p -> propose_gaussian(p, jump_sizes),
8                    initial_pos, 5000)
9 println("Fraction of moves accepted:", accept)
10 size(chain)

```

Fraction of moves accepted:0.3672

Out[31]: (5000, 2)

In [32]:

```

1 # Plot the parameter values in the chain!
2 f = Figure()
3 Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC Samples")
4 scatter!(chain[:,1], chain[:,2], color=:grey)
5 lines!(ellipse_b, ellipse_m, color=:red)
6 f

```

UndefinedVarError: `ellipse\_b` not defined

Stacktrace:

```

[1] top-level scope
      @ In[32]:5

```

Try re-running the MCMC cell above and re-plotting the results. Do the results look the same every time? Does that suggest anything to you about whether the chain has converged after the number of steps we have taken?

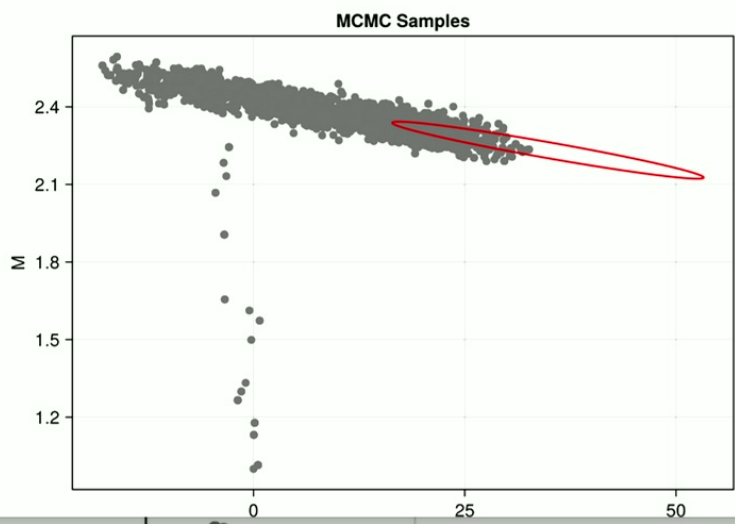
```
8 initial_pos, 5000)
9 println("Fraction of moves accepted:", accept)
10 size(chain)
```

Fraction of moves accepted:0.3672

Out[31]: (5000, 2)

```
In [34]: 1 # Plot the parameter values in the chain!
2 f = Figure()
3 Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC Samples")
4 scatter!(chain[:,1], chain[:,2], color=:grey)
5 lines!(ellipse_b, ellipse_m, color=:red)
6 f
```

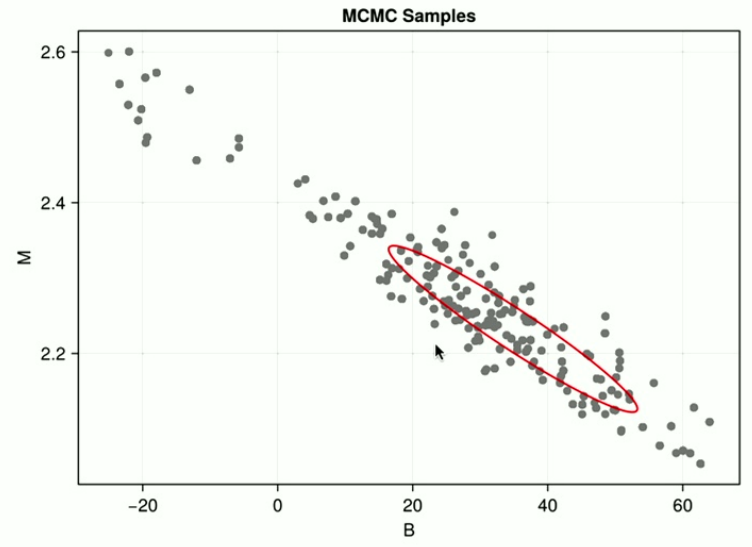
Out[34]:



Out [39]: (5000, 2)

```
In [40]: 1 # Plot the parameter values in the chain!  
2 f = Figure()  
3 Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC Samples")  
4 scatter!(chain[:,1], chain[:,2], color=:grey)  
5 lines!(ellipse_b, ellipse_m, color=:red)  
6 f
```

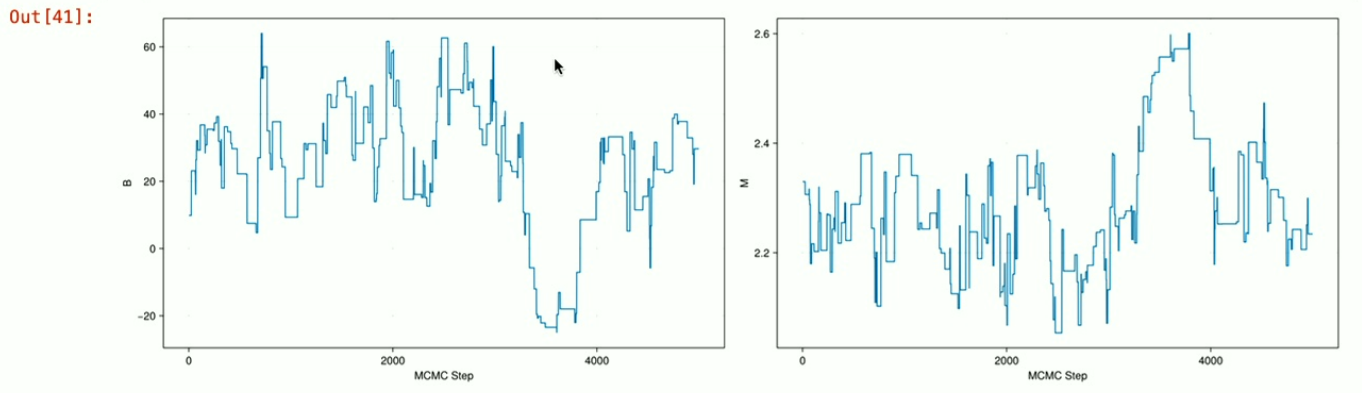
Out [40]:



Try re-running the MCMC cell above and re-plotting the results. Do the results look the same every time? Does that suggest anything to you about whether the chain has converged after the number of steps we have taken?

the chain has converged after the number of steps we have taken?  
Try increasing the number of steps -- do the results look better? How long are you willing to wait?

```
In [41]: 1 # Plot the MCMC chains with respect to sample number
2 f = Figure(size=(1600, 500))
3 Axis(f[1, 1], ylabel="B", xlabel="MCMC Step")
4 lines!(chain[:,1])
5 Axis(f[1, 2], ylabel="M", xlabel="MCMC Step")
6 lines!(chain[:,2])
7 f
```



Looking at these plots of how the "particle" moves through the  $B, M$  parameter space, what do you see? How often does it traverse the whole space? Do you think the step sizes are too big or too small?

```
In [ ]: 1 # Zoom in on the beginning of the chain to see the "burn-in", and repeated values
2 # Plot the MCMC chains with respect to sample number
3 f = Figure(size=(1600, 500))
4 Axis(f[1, 1], ylabel="B", xlabel="MCMC Step")
```

```
7
8     for i in 1:U111, j in 1:U111
9         if i < j
10            continue
11        end
12        ax = Axis(f[i, j], aspect = 1,
13                topspinevisible = false,
14                rightspinevisible = false,)
15        if i == j
16            hist!(x[idxs,i], direction=:y)
17            ax.xlabel = names[i]
18        else
19            scatter!(x[idxs,j], x[idxs,i], markersize=4)
20            ax.xlabel = names[j]
21            ax.ylabel = names[i]
22        end
23    end
24 end;
```

```
In [ ]: 1 cornerplot(chain, ["B", "M"])
```

Try calling the MCMC routine again with step sizes of [0.1, 0.01] and see what the plots look like!

### Tuning MCMC proposal distribution step sizes

If we have two parameters, how do we know which step size we should adjust in order to get a good acceptance ratio?

One approach is to modify our MCMC function so that instead of stepping in both parameters at once, we alternate and step in only one parameter at each step of the algorithm. We could try to write a fancy general version of that, but instead let's just copy-paste the MCMC routine and customize it for this task! Once we've selected good step sizes we can go back to the regular version.

```
In [ ]: 1 function mcmc_cyclic(logprob_func,
2                             propose_func,
3                             initial_pos, nsteps)
4         """"
5         This is a variation on the "vanilla" MCMC algorithm, where we change the proposal function
```

### Tuning MCMC proposal distribution step sizes

If we have two parameters, how do we know which step size we should adjust in order to get a good acceptance ratio?

One approach is to modify our MCMC function so that instead of stepping in both parameters at once, we alternate and step in only one parameter at each step of the algorithm. We could try to write a fancy general version of that, but instead let's just copy-paste the MCMC routine and customize it for this task! Once we've selected good step sizes we can go back to the regular version.

```
In [ ]: 1 function mcmc_cyclic(logprob_func,
2                 propose_func,
3                 initial_pos, nsteps)
4         """
5         This is a variation on the "vanilla" MCMC algorithm, where we change the proposal function
6         to modify only a single parameter in each step of the MCMC. We record the acceptance ratio
7         separately for each parameter.
8         """
9         p = initial_pos
10        logprob = logprob_func(p)
11        chain = zeros(Float64, (nsteps, length(p)))
12        naccept = zeros(Int, length(p))
13        for i = 1:nsteps
14            # propose a new position in parameter space
15            p_jump = propose_func(p)
16            # BUT, only copy one element (cycle through the elements);
17            # keep the rest the same!
18            # The index of the parameter to change:
19            j = 1 + ((i-1) % length(p))
20            p_new = copy(p)
21            p_new[j] = p_jump[j]
22            # compute probability at new position
23            logprob_new = logprob_func(p_new)
24            # decide whether to jump to the new position
25            if exp(logprob_new - logprob) > rand()
26                p = p_new
27                logprob = logprob_new
28                naccept[j] += 1
29        end
```

Firefox File Edit View History Bookmarks Tools Window Help 10:28 MST 11:04:32:20 Fri Jan 31 12:28 PM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Julia 1.10.5

```

25         if exp(logprob_new - logprob) > rand()
26             p = p_new
27             logprob = logprob_new
28             naccept[j] += 1
29         end
30         # save the position
31         chain[i,:] = p
32     end
33     # Since we cycle through the parameters, the number of steps per parameter
34     # is (approximately) (nsteps) / (number of parameters).
35     return chain, naccept/(nsteps/length(p))
36 end;

```

In [49]:

```

1 # initial B,M
2 initial_pos = [0., 1.0]
3 # proposal distribution: jump sizes for B,M
4 ##### CHANGE THESE -- we were using [1, 0.1] before. Play around with these values until you get
5 ##### acceptance ratios of about 0.5 per coordinate!
6 jump_sizes = [4, 0.1]
7 # Run MCMC!
8 chain,accept = mcmc_cyclic(
9     p -> log_posterior_one(p, x, y, yerr),
10    p -> propose_gaussian(p, jump_sizes),
11    initial_pos, 5000)
12 println("Fraction of moves accepted (for B & M, respective):", accept)
13 size(chain)

```

Fraction of moves accepted (for B & M, respective):[0.7788, 0.3508]

Out[49]: (5000, 2)

Try modifying the `jump_sizes` values above until you get acceptance ratios of about 0.5 for each parameter. Recall that proposing smaller jumps should result in a larger acceptance ratio.

In [ ]:

```

1 # Now let's plug those *jump_sizes* into our "vanilla" MCMC. Since we are now jumping in both parameters
2 # at once, the acceptance ratio will be a bit smaller.
3 # initial B,M
4 initial_pos = [0., 1.0]

```



```
5 ##### acceptance ratios of about 0.5 per coordinate!
6 jump_sizes = [15., 0.1]
7 # Run MCMC!
8 chain,accept = mcmc_cyclic(
9     p -> log_posterior_one(p, x, y, yerr),
10    p -> propose_gaussian(p, jump_sizes),
11    initial_pos, 5000)
12 println("Fraction of moves accepted (for B & M, respective):", accept)
13 size(chain)
```

Fraction of moves accepted (for B & M, respective):[0.4, 0.3404]

Out[51]: (5000, 2)

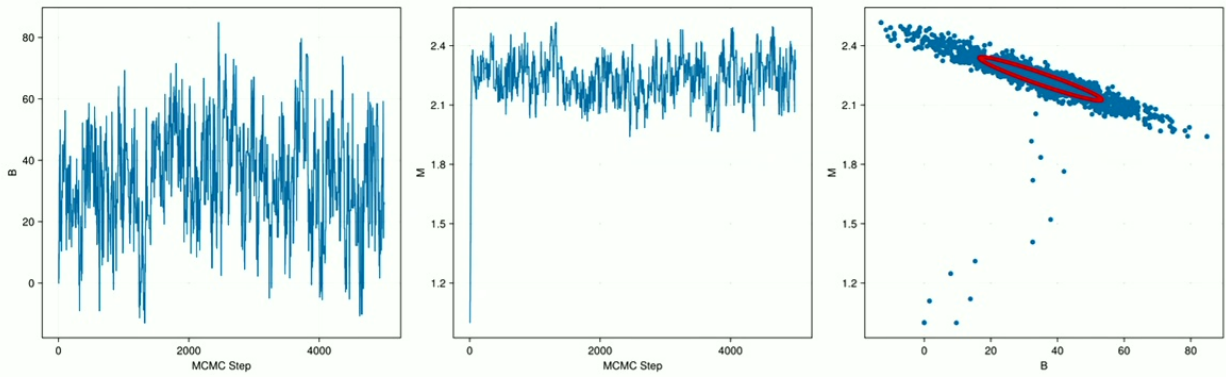
Try modifying the `jump_sizes` values above until you get acceptance ratios of about 0.5 for each parameter. Recall that proposing smaller jumps should result in a larger acceptance ratio.

```
In [55]: 1 # Now let's plug those *jump_sizes* into our "vanilla" MCMC. Since we are now jumping in both parameters
2 # at once, the acceptance ratio will be a bit smaller.
3 # initial B,M
4 initial_pos = [0., 1.0]
5 # proposal distribution: jump sizes for B,M
6 jump_sizes = [15., 0.1]
7 # Run MCMC!
8 chain,accept = mcmc(
9     p -> log_posterior_one(p, x, y, yerr),
10    p -> propose_gaussian(p, jump_sizes),
11    log_posterior_one,
12    propose_gaussian,
13    initial_pos, 5000)
14 println("Fraction of moves accepted:", accept)
15 size(chain)
```

MethodError: no method matching `log_posterior_one(::Vector{Float64})`

Closest candidates are:  
`log_posterior_one(::Any, ::Any, ::Any, ::Any)`  
@ Main In[30]:1

Out [57]:



That looks way better! Our samples are traversing the state space many times.

Let's also look at the resulting corner plot.

```
In [ ]: 1 nburn = 1000
        2 cornerplot(chain, ["B", "M"])
```

### Extensions

- If you are interested, try extending the model from a linear model to a quadratic model. That is, switch to  $y_{\text{pred}} = b + mx + qx^2$ . You will need to write a new `log_likelihood_quadratic` function that expects three parameters. Run MCMC on that model, plot the results, and show the corner plots. Does it look like the model "needs" the quadratic term?
- What if the  $\sigma$  values are estimated incorrectly? (Eg, if your experimenter friends overlooked or mis-estimated some source of error in their data collection!) Try increasing or decreasing the `yerr` values by a factor of 2 and re-make the plots. How do the constraints on  $B$  and  $M$  change? How does the visual quality of the fit change?

Firefox File Edit View History Bookmarks Tools Window Help 10:31 MST 11:04:34:45 Fri Jan 31 12:31 PM

MCMC localhost:8890/notebooks/PSI-Numerical-Methods-2025/session-17/MCMC.ipynb# 120% Search

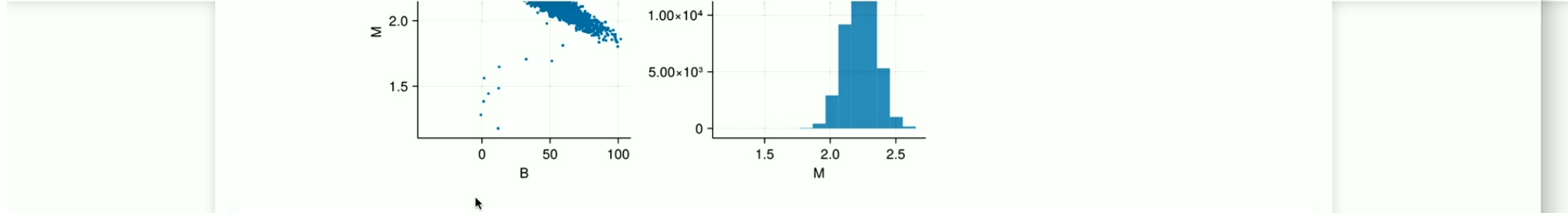
jupyter MCMC Last Checkpoint: 18 hours ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Julia 1.10.5

```
re's has been deprecated. Use `Figure(; size = ...` or `Scene(; size = ...)` instead, which better reflects that this is a unitless size and not a pixel resolution. The key could also come from `set_theme!` calls or related theming functions.  
@ Makie ~/.julia/packages/Makie/Y3ABD/src/scenes.jl:238
```

Out [60]:

The figure displays three plots related to the variables B and M. The top plot is a histogram of B, showing a distribution centered around 40 with a peak frequency of approximately 10,000. The bottom-left plot is a scatter plot of M versus B, showing a strong negative linear correlation between the two variables. The bottom-right plot is a histogram of M, showing a distribution centered around 2.2 with a peak frequency of approximately 1.5 x 10^4.



### Extensions

- If you are interested, try extending the model from a linear model to a quadratic model. That is, switch to  $y_{\text{pred}} = b + mx + qx^2$ . You will need to write a new `log_likelihood_quadratic` function that expects three parameters. Run MCMC on that model, plot the results, and show the corner plots. Does it look like the model "needs" the quadratic term?
- What if the  $\sigma$  values are estimated incorrectly? (Eg, if your experimenter friends overlooked or mis-estimated some source of error in their data collection!) Try increasing or decreasing the `yer_r` values by a factor of 2 and re-make the plots. How do the constraints on  $B$  and  $M$  change? How does the visual quality of the fit change?
- We found jump sizes for  $B$  and  $M$  the led to okay acceptance ratios, but we are still taking jumps independently in the two variables, while we can clearly see that the variables are correlated. Can you come up with a new `propose_func` that proposes jumps drawn from a Gaussian with appropriate covariance? (you can check out where I sample from the inverse-Hessian ellipse, above, for how to sample from a multivariate Gaussian distribution given its covariance).
- We used "uninformative" priors on  $B$  and  $M$ . Try changing that -- for example, try placing a Gaussian (log-)prior on one of the parameters, and see how that affects your samplings. How strong do you have to make the prior for it to have a significant effect on your results?

I hope this has been an interesting glimpse into Markov Chain Monte Carlo in practice!